Chapter 11 Reliability engineering:

11.1 Availability and reliability

11.2 Reliability requirements

11.3 Fault-tolerant architectures

11.4 Programming for reliability

11.5 Reliability measurement

Note: Images excluded due to time constraints

11.1 Availability and Reliability

Overview

- **Availability** and **reliability** are aspects of system performance, often expressed as probabilities.
- Availability is about whether a service is up and running, while reliability is about the correctness of the service output.

Core Definitions

1. **Reliability**: The probability of failure-free operation over a specified time, in a given environment, for a specific purpose.
2. **Availability**: The probability that a system, at a point in time, will be operational and able to deliver the requested services.

Factors Influencing Reliability

- **System Environment**: Reliability varies depending on how and where the system is used.
- **User Behavior**: Different users and their usage patterns can affect the perception of reliability.

Problems with Reliability Definitions

1. **Incomplete Specifications**: Software specs may be incomplete or incorrect, making it challenging to define "failure."
2. **User Expectations**: Users may have different expectations from what is specified, leading to different perceptions of failures.

Input/Output Mapping

- A system can be thought of as a mapping between a set of inputs and outputs.
- Reliability depends on the inputs that lead to erroneous outputs; not all inputs lead to system failure.

Availability Factors

- Availability depends not just on the number of system failures but also on the **time needed for repairs**.
- **Timing of Failures**: When the system fails is crucial; off-hours failures may be less disruptive.

Reliability vs. Availability

- Sometimes, one is more important than the other depending on the system requirements.
- Example: In a telephone exchange system, availability is more important because failures are quickly recoverable.

Understanding the nuanced differences and implications of reliability and availability can help in the effective design and maintenance of systems.

11.2 Reliability Requirements

Overview

- The section discusses the importance of **system dependability** in relation to **reliability requirements**.
- Reliability requirements are crucial for ensuring that a system operates safely and dependably, even in rare and unexpected situations.

Case Study: Plane Accident

- A 1993 plane accident highlighted the need for comprehensive reliability requirements.
- The braking system software functioned according to its specification, but the specification was incomplete and did not account for rare situations.

Types of Reliability Requirements

1. **Functional Requirements**: These define the checking and recovery facilities that should be included in the system. They also include features for protection against system failures and external attacks.
2. **Nonfunctional Requirements**: These define the required reliability and availability of the system.

Factors Affecting Reliability

- The overall reliability of a system depends on **hardware reliability**, **software reliability**, and the **reliability of system operators**.
- The system software must consider these aspects in its reliability requirements.

Availability Metrics

- **0.9**: System is available for 90% of the time, leading to 144 minutes of unavailability in a 24-hour period.
- **0.99**: System is unavailable for 14.4 minutes in a 24-hour period.
- **0.999**: System is unavailable for 84 seconds in a 24-hour period.
- **0.9999**: System is unavailable for 8.4 seconds in a 24-hour period, roughly equivalent to one minute per week.

Importance of Detail

- Attention to detail in deriving system requirements is essential for ensuring system dependability.

Reliability requirements, both functional and nonfunctional, play a pivotal role in ensuring that systems operate as expected even under rare or extreme conditions.

11.2.1 Reliability Metrics

Overview

- The section delves into how **reliability** can be specified and measured through various **metrics**.
- These metrics help in understanding the likelihood of system failures in different operating environments.

Types of Reliability Metrics

1. **Probability of Failure on Demand (POFOD)**: Defines the probability that a demand for service will result in a system failure.
    - For instance, POFOD = 0.001 means a 1/1000 chance of failure upon demand.
    - Suitable for systems where failure on demand could result in catastrophic outcomes, such as a chemical reactor's protection system.
2. **Rate of Occurrence of Failures (ROCOF)**: Indicates the probable number of system failures within a given time period or number of executions.
    - The reciprocal of ROCOF is the Mean Time To Failure (MTTF), another metric for reliability.
    - ROCOF is useful for systems where demands are made regularly. For example, in a high-transaction system, a ROCOF of 10 failures per day could be acceptable.
3. **Availability (AVAIL)**: Specifies the probability that a system will be operational when a service demand is made.
    - For example, an AVAIL of 0.9999 means the system is available for 99.99% of the operating time.

Application of Metrics

- **POFOD**: Best suited for systems where failure could be catastrophic, even if demands are infrequent.
- **ROCOF**: Applicable when demands on the system are frequent. It could be expressed as failures per day or failures per 1000 transactions.

- **MTTF (Mean Time To Failures)**: Useful when the absolute time between failures is significant. For example, in long-transaction systems like computer-aided design, MTTF should be much longer than the average user session to minimize work loss.

Importance of Context

- The choice of metric often depends on the **nature of the system** and its **operating environment**.

The metrics serve as crucial tools for understanding and specifying system reliability, each fitting different scenarios and demands.

11.2.2 Nonfunctional Reliability Requirements

Overview

- Discusses the importance of specifying **nonfunctional reliability requirements** using metrics like **POFOD**, **ROCOF**, and **AVAIL**.
- Quantitative reliability specification is becoming increasingly crucial, especially for 24/7 service demands.

Importance of Quantitative Specification

1. **Clarification for Stakeholders**: Helps in understanding the real needs and costs associated with different levels of system failure.
2. **Testing Endpoint**: Provides a basis for deciding when to cease testing, i.e., when the system reaches the required reliability level.
3. **Design Strategy Assessment**: Aids in evaluating different design approaches to improve system reliability.
4. **Regulatory Approval**: Essential for system certification, especially for safety-critical systems.

Guidelines for Specifying Reliability

1. **Type of Failure**: Differentiate between high-cost failures and less serious ones, assigning lower probabilities to the former.
2. **System Service Types**: Specify requirements for different services; critical services should have higher reliability.
3. **Need for High Reliability**: Evaluate if high reliability is genuinely needed, especially if error-detection and correction mechanisms are in place.

Real-world Examples

- **Bank ATM Systems**: The focus is more on availability rather than transaction correctness, as most transaction errors can be corrected later.
- **Insulin Pump Control Software**: Considers two types of failure - transient and permanent. Each has a different acceptable POFOD value depending on the criticality and frequency of the operation.

Over-specification Risks

- **Overspecification** of reliability can exponentially increase development costs. Hence, it's crucial to specify the actual needed reliability, especially when different parts of the system have varied criticality.

By carefully specifying nonfunctional reliability requirements, organizations can better align their systems with stakeholder needs and regulatory standards while avoiding unnecessary costs.

11.2.3 Functional Reliability Specification

Overview

- Discusses the necessity of **functional reliability requirements** that guide how a system should offer fault avoidance, detection, and tolerance.
- These requirements are derived after analyzing nonfunctional requirements and assessing associated risks.

Types of Functional Reliability Requirements

1. **Checking Requirements**: These identify checks on system inputs to detect incorrect or out-of-range inputs before processing.
2. **Recovery Requirements**: Aimed at enabling system recovery post-failure. These often involve maintaining backup copies of the system and data and outlining restoration procedures.
3. **Redundancy Requirements**: Specify additional, redundant system features to ensure that a single component failure doesn't result in a total service loss.
4. **Process Requirements**: Focus on fault-avoidance by ensuring best practices in the development process, aimed at reducing the number of system faults.

Deriving Functional Reliability Requirements

- There are no straightforward rules for deriving these requirements.
- Organizations with experience in developing critical systems usually possess institutional knowledge that can be reused for specifying reliability requirements across different types of systems.

By defining functional reliability requirements, organizations can better manage the risks associated with system faults and ensure a higher level of system dependability.

11.3 Fault-tolerant Architectures

Overview

- **Fault tolerance** is a crucial aspect of dependability, allowing systems to continue operation even when faults occur.
- This is particularly important in systems that are safety or security critical, such as aircraft systems, telecommunication systems, and critical command and control systems.

Characteristics of Fault-tolerant Systems

- Systems are designed with **redundant and diverse hardware and software** to detect and correct erroneous states.

Replicated Servers

- One common method is using **replicated servers** where multiple servers carry out the same task.
- A server management component routes each request to a specific server and keeps track of server responses.
- In case of server failure, unprocessed requests are rerouted to other functioning servers.

Limitations of Replicated Servers

- While they provide **redundancy**, they usually do not offer **diversity**.
- The hardware is often identical, and servers run the same version of software.
- This approach can handle hardware failures and localized software failures but not design problems affecting all versions of the software.

Diverse Software and Hardware

- To cope with software design failures, systems need to incorporate **diverse software and hardware**.

Architectural Patterns

- Various architectural patterns exist for fault-tolerant systems, although the section indicates that these will be described in following sections.

By incorporating fault tolerance into the architecture, systems can enhance their dependability, especially in safety and security critical applications.

11.3.1 Protection Systems

Overview

- **Protection systems** are specialized systems associated with control systems for processes or equipment.
- These systems autonomously monitor the environment and the controlled equipment, activating when the primary system fails to address a detected issue.

Functionality and Architecture

- Protection systems are designed to include **critical functionality** needed to transition the system from an unsafe to a safe state.
- They often have **redundant actuators** and two sets of sensors, one for normal system monitoring and another specifically for the protection system.

Relationship with Controlled System

- They work in parallel with a **controlled system**, such as a chemical manufacturing process or an equipment control system like driverless trains.
- If the controlled system fails to address an issue, the protection system issues commands to actuators to either shut down the system or engage other safety measures.

Advantages

1. **Simplicity**: The software for the protection system is simpler than the software controlling the process.
2. **High Reliability**: Due to its simplicity, more effort can be invested in **fault avoidance and detection**, aiming for a very low probability of failure on demand (e.g., 0.001).

Protection systems are an example of a more general **fault-tolerant architecture** where a principal system is supported by a smaller and simpler backup system focused on essential functions. They aim for high reliability and are especially valuable where system safety is critical.

11.3.2 Self-monitoring Architectures

Overview

- **Self-monitoring architectures** are designed for systems to monitor their own operations and take corrective actions if discrepancies are detected.
- They are particularly useful in contexts where **reliability** is more critical than **availability**, such as medical treatment and diagnostic systems.

Architecture Features

1. **Multiple Channels**: Computations are carried out on separate channels, and their outputs are compared.
    - If outputs are identical and available simultaneously, the system is assumed to be operating correctly.
    - If outputs differ, a failure is assumed, triggering a failure exception.
2. **Hardware Diversity**: Each channel should use diverse hardware, potentially from different manufacturers, to reduce the likelihood of common design faults.
3. **Software Diversity**: Different channels should also use diverse software to mitigate the risk of simultaneous software errors.

High-Availability Contexts

- In scenarios requiring high availability, multiple self-monitoring systems operate in parallel.
- A **switching unit** detects faults and selects a consistent result from one of the systems.

Case Study: Airbus 340 Series

- The Airbus 340 flight control system uses **five self-checking computers**.
- This architecture allows for up to four computers to fail without loss of aircraft control.
- Various strategies are used to achieve hardware and software diversity, including using different processors, chipsets, and software developed by different teams.

The architecture aims to drastically reduce the probability of common failures across different channels, making it highly reliable for critical operations.

11.3.3 N-version Programming

Overview

- **N-version Programming** is an approach that leverages software redundancy and diversity to achieve fault tolerance.
- Derived from hardware-based **Triple Modular Redundancy (TMR)**, it has been used in critical systems like railway signaling, aircraft, and reactor protection systems.

Mechanism and Components

1. **Multiple Versions**: Multiple diverse versions of a software system are developed based on a common specification.
    - Each version is executed on a separate computer.
2. **Output Comparison**: The outputs from these versions are compared using a voting system.
    - Inconsistent or tardy outputs are rejected.
    - At least three versions should be available to ensure consistency in the case of a single failure.
3. **Fault Manager**: An optional component that may try to repair a faulty version automatically, or reconfigure the system to exclude it.

Assumptions and Requirements

- **Independent Failures**: Assumes that components (software versions) are likely to fail independently.
- **Low Probability of Simultaneous Failure**: Assumes that there is a low chance of simultaneous component failure in all versions.

Advantages and Limitations

- **Cost-Effectiveness**: May be less expensive than self-monitoring architectures for systems requiring high availability.
- **High Development Costs**: Requires multiple development teams, leading to high software development costs.
- **Specialized Use-Case**: Primarily used in systems where traditional protection systems are impractical for guarding against safety-critical failures.

The approach aims to mitigate the risks associated with common design faults by ensuring that each version is independently developed, thereby reducing the likelihood of simultaneous failures.

11.3.4 Software Diversity

Overview

- **Software Diversity** is crucial for fault-tolerant architectures, aiming to ensure that different implementations of the same specification will not fail simultaneously.

- The goal is to maximize differences between versions to prevent common failures, especially in safety- and mission-critical systems.

Strategies for Achieving Diversity

1. **Different Design Methods**: Require teams to use varying design paradigms, such as object-oriented and function-oriented designs.
2. **Different Programming Languages**: Stipulate the use of various languages like Ada, C++, and Java for different software versions.
3. **Different Development Tools**: Mandate the use of diverse tools and development environments.
4. **Different Algorithms**: Require the use of varying algorithms in certain parts of the implementation, although this may conflict with performance requirements.

Theoretical vs. Practical Diversity

- **Ideal Scenario**: In a perfect world, each version should have zero dependencies on the others, maximizing the overall system reliability.
- **Real-World Constraints**: Complete independence is practically impossible due to cultural, educational, and specification-related commonalities among development teams.

Factors Affecting Common Failures

1. **Cultural and Educational Background**: Teams may share a similar background, leading to common misunderstandings or mistakes.
2. **Faulty Requirements**: If the requirements are incorrect or ambiguous, this will be reflected in each version.
3. **Ambiguous Specifications**: Different teams may misinterpret vague specifications in the same manner.

Reliability Analysis

- A three-channel system was found to be between 5 and 9 times more reliable than a single-channel system.
- However, it's unclear if the increased reliability justifies the additional development costs, except in cases where failure costs are extremely high.

Software diversity aims to mitigate common failures but faces challenges due to human factors and the complexities of complete independence. It is most applicable in systems where the costs of failure are too high to risk a single-point failure.

11.4 Programming for Reliability

Overview

- **Programming for Reliability** focuses on language-independent good practices that enhance system reliability and security.

Dependable Programming Guidelines

1. **Control the Visibility of Information**: Adopt the "need to know" principle. Limit access to variables and data structures to only those components that need them.
2. **Check All Inputs for Validity**: Inputs should be validated as soon as they are read. Types of checks include:
    - Range checks
    - Size checks
    - Representation checks
    - Reasonableness checks
3. **Provide a Handler for All Exceptions**: Handle all possible exceptions to prevent system failure.
    - Exception-handling approaches may signal to a higher-level component, carry out alternative processing, or pass control to the language runtime.
4. **Minimize Use of Error-Prone Constructs**: Avoid using constructs that are known to be error-prone like floating-point numbers and dynamic storage allocation.
5. **Provide Restart Capabilities**: Implement checkpoints in both short and long transaction systems to allow the system to restart from the most recent valid state.
6. **Check Array Bounds**: Always include checks for array index boundaries, especially in languages that do not automatically do this.
7. **Include Timeouts for External Calls**: In distributed systems, always include a timeout when waiting for a response from an external component.
8. **Name All Constants**: Constants that represent real-world values should be named for easier maintenance and to reduce errors.

Data Structures and Information Hiding

- Implementing data structures as **Abstract Data Types** helps to hide their internal structure, making it easier to change the data representation without affecting other components.

Input Validation

- Input validation is essential for both preventing user errors and enhancing security.

Exception Handling

- Built-in exception-handling constructs in languages like Java, C++, and Python can be used to manage errors and unexpected events.

Error-Prone Constructs

- Some constructs are inherently prone to errors. While they can't always be avoided, their usage should be minimized and controlled within abstract data types.

Restart Capabilities

- Providing restart capabilities is crucial for both short transactions like e-commerce and long transactions like CAD systems.

Timeout and External Calls

- Timeouts are essential in distributed systems to assume failure of non-responsive components and to take appropriate action.

By adhering to these guidelines and practices, the reliability and fault-tolerance of a software system can be significantly improved.

11.5 Reliability Measurement

Overview

- **Reliability Measurement** involves collecting data on system operation to assess its dependability. Metrics like **POFOD** (Probability of Failure on Demand), **ROCOF** (Rate of Occurrence of Failure), and **MTTF** (Mean Time To Failure) are crucial.

Data Required for Reliability Measurement

1. **Number of System Failures**: Used to measure POFOD.
2. **Time or Transactions Between Failures**: Used to measure ROCOF and MTTF.
3. **Repair or Restart Time**: Essential for measuring availability.

Time Units in Metrics

- **Calendar Time**: Used for systems in continuous operation, like monitoring systems.
- **Number of Transactions**: Suitable for systems like ATMs or reservation systems with variable loads.

Statistical Testing for Reliability

1. **Study Existing Systems**: Define an operational profile based on how similar systems are used.
2. **Construct Test Data**: Create test data reflecting the operational profile, often using a test data generator.
3. **Apply Tests**: Log the number, type, and time of failures.
4. **Compute Reliability**: After observing a statistically significant number of failures, compute the software reliability metrics.

Challenges in Reliability Measurement

1. **Operational Profile Uncertainty**: Profiles based on other systems may not accurately reflect real usage.
2. **High Costs of Test Data Generation**: Large volumes of test data can be expensive to generate.
3. **Statistical Uncertainty**: Difficult to generate a significant number of failures in highly reliable systems.
4. **Recognizing Failure**: Ambiguities in natural language specifications can make it hard to identify failures.

Test Data Generation

- **Automated test data generation** is the most efficient way to produce large datasets but may be challenging for interactive systems.

Fault Injection

- Deliberate injection of errors to study their conversion into faults and failures. Useful for assessing the effectiveness of defect testing but limited in predicting faults stemming from requirements or design problems.

Reliability Growth Modeling

- A conceptual model that evolves into a mathematical model to predict how system reliability changes over time during testing and debugging.

By considering these aspects and metrics, a more complete and accurate understanding of a system's reliability can be achieved.

11.5.1 Operational Profiles

Overview

- **Operational Profiles** represent the expected usage patterns of a software system. They specify classes of input and the probability of their occurrence.

Operational Profiles in Established Systems

- Easier to define for systems replacing existing automated systems. The profile reflects existing usage patterns with adjustments for new functionalities.
- Example: In **telecommunication systems**, companies know the call patterns, making operational profile development straightforward.

Complexity in Profile Definition

1. **Highly Probable Inputs**: A few classes of inputs usually have the highest likelihood of being generated.
2. **Uncommon Inputs**: Many classes exist where inputs are rare but still possible.

Development Time

- For systems with historical data, like telecommunication systems, operational profile development is quicker (e.g., 1 person-month for a 15 person-year project).

Challenges in New and Innovative Systems

- Difficult to anticipate usage patterns due to a variety of users and lack of historical data.
- Users may interact with systems in unanticipated ways, making it challenging to develop an accurate operational profile.

Limitations and Pitfalls

1. **Variability in User Behavior**: Different users have different patterns, making it hard to capture all in one operational profile.

2. **Changing User Behavior**: Over time, as users become more familiar with a system, their usage patterns evolve, making initial operational profiles outdated.

Impact on Reliability Measurement

- Using an incorrect or outdated operational profile leads to unreliable reliability measurements.

By understanding these elements, one can better appreciate the complexities and limitations involved in developing operational profiles for different types of systems.

Summary

- Software reliability can be achieved by avoiding the introduction of faults, by detecting and removing faults before system deployment, and by including fault-tolerance facilities that allow the system to remain operational after a fault has caused a system failure.
- Reliability requirements can be defined quantitatively in the system requirements specification.
- Reliability metrics include probability of failure on demand (POFOD), rate of occurrence of failure (ROCOF), and availability (AVAIL).
- Functional reliability requirements are requirements for system functionality, such as checking and redundancy requirements, which help the system meet its non-functional reliability requirements.
- Dependable system architectures are system architectures that are designed for fault tolerance.
- A number of architectural styles support fault tolerance, including protection systems, selfmonitoring architectures, and N-version programming.
- Software diversity is difficult to achieve because it is practically impossible to ensure that each version of the software is truly independent.
- Dependable programming relies on including redundancy in a program as checks on the validity of inputs and the values of program variables.
- Statistical testing is used to estimate software reliability. It relies on testing the system with test data that matches an operational profile, which reflects the distribution of inputs to the software when it is in use.